Article

Intelligent Visual Representation for Java Code Data in the Field of Software Engineering Based on Remote Sensing Techniques

Dian Li [†], **Weidong Wang** [*,†,‡] and **Yang Zhao**

Department of Software Engineering, Beijing University of Technology, Beijing 100124, China; iamlidian@emails.bjut.edu.cn (D.L.); zhaoyang@emails.bjut.edu.cn (Y.Z.)
* Correspondence: wangweidong@bjut.edu.cn
† These authors contributed equally to this work.
‡ Current address: Room 816, Software Building, Beijing University of Technology, Chaoyang District, Beijing 100124, China.

Abstract: In the field of software engineering, large and complex code bases may lead to some burden of understanding their structure and meaning for developers. To reduce the burden on developers, we consider a code base visualization method to visually express the meaning of code bases. Inspired by remote sensing imagery, we employ graphical representations to illustrate the semantic connections within Java code bases, aiming to help developers understand its meaning and logic. This approach is segmented into three distinct levels of analysis. First, at the project-level, we visualize Java projects by portraying each file as an element within a code forest, offering a broad overview of the project's structure. This macro-view perspective aids in swiftly grasping the project's layout and hierarchy. Second, at the file-level, we concentrate on individual files, using visualization techniques to highlight their unique attributes and complexities. This perspective enables a deeper understanding of each file's structure and its role within the larger project. Finally, at the component-level, our focus shifts to the detailed analysis of Java methods and classes. We examine these components for complexity and other specific characteristics, providing insights that are crucial for the optimization of code and the enhancement of software quality. By integrating remote sensing technology, our method offers software engineers deeper insights into code quality, significantly enhancing the software development lifecycle and its outcomes.

Keywords: remote sensing technology; software engineering; code visualization; remote sensing imagery; java code analysis

check for updates

Citation: Li, D.; Wang, W.; Zhao, Y. Intelligent Visual Representation for Java Code Data in the Field of Software Engineering Based on Remote Sensing Techniques. *Electronics* **2023**, *12*, 5009. https://doi.org/10.3390/electronics12245009

Academic Editor: Silvia Liberata Ullo

Received: 20 November 2023
Revised: 4 December 2023
Accepted: 5 December 2023
Published: 14 December 2023

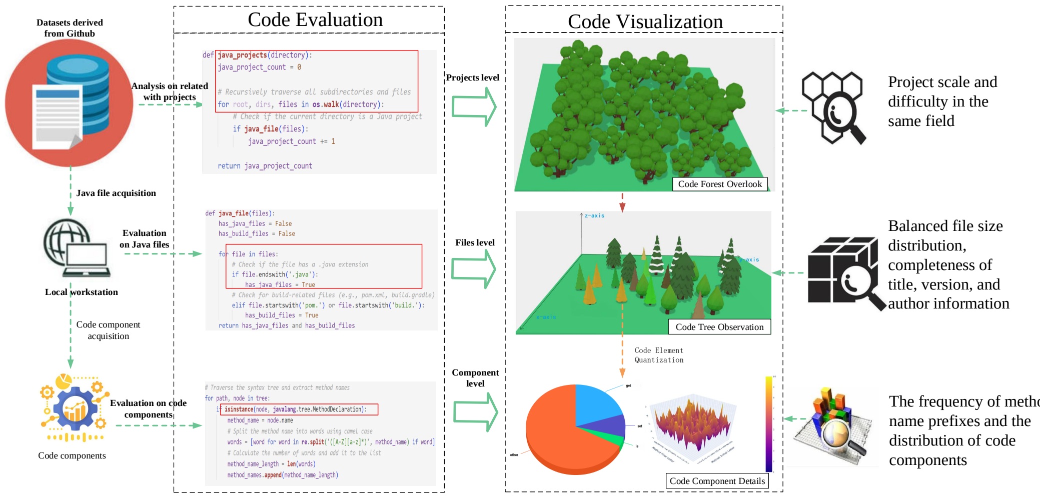
1. Introduction

In the field of software engineering, deeply understanding the structure and meaning of codebases is a crucial but troublesome task for programmers [1,2]. This not only adds an extra burden on them but also increases the monotony of programming work [3,4]. While existing methods like dependency graphs, UML (Unified Modeling Language) views and code annotations provide a foundation for understanding code structure [5,6], they are limited in terms of intuitiveness and comprehensibility. We specifically selected the Java code analysis platform due to its widespread adoption in the software engineering community, its comprehensive library ecosystem, and robust support network, making it highly suitable for our innovative visualization approach.

Traditional remote sensing methods, known for their proficiency in data analysis and visualization, are adept at processing and interpreting various data types. These methods have found extensive application in diverse fields such as Earth science, environmental monitoring [7–9], and agriculture [10]. In software engineering, applying these methods to code data opens new avenues for extracting insights about code quality, complexity, and structure. This paper concentrates on merging data analysis and visualization techniques from remote sensing with code data in software engineering [11]. By preprocessing and transforming code data, and utilizing intelligent evaluation techniques and data mining algorithms [12,13], our method offers a multi-perspective analysis that uncovers crucial information about code complexity, structure, and potential issues.

As illustrated in Figure 1, the multi-perspective analysis offered by our method encompasses three distinct views: the project view, the file view, and the code component view. These perspectives collectively serve to enhance software engineers' understanding and analysis capabilities of Java codebases. First, the project view, also known as the "code forest", offers a macro-view overview of the project structure. In this view, each Java file is innovatively depicted as a tree, creating a metaphorical forest. This visualization is not only intuitive but also insightful, enabling programmers to quickly grasp the overall architecture and interrelations of project components. Observing the density and distribution of the "code forest" allows programmers to easily identify important modules and potential complex areas in the project. The file view delves deeper into individual file attributes, encouraging programmers to engage with file-view complexities. This view assists programmers in understanding how each file interacts with other parts of the project and how to optimize these files to enhance the overall code quality. Finally, the code component view offers a detailed examination of individual Java methods, considering aspects like complexity and method-specific characteristics, thus enabling programmers to better understand the functionality of the code and areas for potential improvement [14]. The contributions of this paper are summarized in the following:

- We introduce a method that utilizes remote sensing imaging techniques to visualize Java codebases. This method not only simplifies the understanding of complex code structures but also makes it more intuitive and engaging for programmers.
- We present a multi-perspective analysis (project-, file-, and component-level) that provides comprehensive insights into code complexity, structure, and potential issues to assist in the analysis of code.


Datasets derived from Github
Analysis on related with projects
Java file acquisition
Evaluation on Java files
Local workstation
Code component acquisition
Evaluation on code components
Code components
Code Evaluation
Projects level
Files level
Component level
Code Visualization
Code Forest Overlook
Code Tree Observation
Code Element Quantization
Code Component Details
Project scale and difficulty in the same field
Balanced file size distribution, completeness of title, version, and author information
The frequency of method name prefixes and the distribution of code components

```
def java_projects(directory):
    java_project_count = 0

    # Recursively traverse all subdirectories and files
    for root, dirs, files in os.walk(directory):
        # Check if the current directory is a Java project
        if java_file(files):
            java_project_count += 1

    return java_project_count
```

```
def java_file(files):
    has_java_files = False
    has_build_files = False

    for file in files:
        # Check if the file has a .java extension
        if file.endswith('.java'):
            has_java_files = True
        # Check for build-related files (e.g., pom.xml, build.gradle)
        elif file.startswith('pom.') or file.startswith('build.'):
            has_build_files = True
    return has_java_files and has_build_files
```

```
# Traverse the syntax tree and extract method names
for path, node in tree:
    if isinstance(node, javalang.tree.MethodDeclaration):
        method_name = node.name
        # Split the method name into words using camel case
        words = [word for word in re.split('([A-Z][a-z]*)', method_name) if word]
        # Calculate the number of words and add it to the list
        method_name_length = len(words)
        method_names.append(method_name_length)
```

Figure 1. Multi-perspective visualization for enhanced code understanding.

Based on this context, our study introduces a novel method of codebase visualization, designed to make the structure and meaning of code more apparent through intuitive and visual means [15]. This method, which employs remote sensing imaging techniques, effectively reveals and illustrates the semantic connections within Java codebases. In this way, it assists developers in swiftly and accurately extracting information, simplifying the complexities of the code, and reducing the monotony typically associated with programming tasks.

Our experimental results show that integrating three-dimensional terrain mapping techniques from remote sensing into software engineering provides a novel, intuitive method for code visualization. This method significantly improves the comprehensibility and maintainability of software systems and brings new insights into the software development process. The relevant source code can be found at https://github.com/lidiancracy/Java_3D. The rest of the paper is organized as follows: Section 2 reviews related work, situating our study within the broader literature. Section 3 provides a detailed description of our proposed method, including the design of evaluation metrics [16]. Section 4 discusses the practicality of our method, highlighting its accuracy, efficiency, and user satisfaction. Finally, Section 5 summarizes our contributions and explores future research directions in this exciting area.

2. Related Work

The integration of remote sensing techniques into engineering has garnered increasing attention in recent years, a trend evidenced by a surge in research efforts [17–19]. This emerging field reflects a growing awareness of the value that such techniques can bring to the complex world of development.

Early applications of remote sensing in environmental and geological sciences, as exemplified by the use of airborne laser scanning to map tree crowns, have significantly advanced our understanding of natural landscapes and proved invaluable for forestry management [20,21]. This pioneering method underscores the precision and versatility of remote sensing techniques. Similarly, advances in 3D radiative transfer modeling, integral to the field of remote sensing, have shed light on the complexities and computational demands involved in accurately simulating natural processes [22]. This progress paves the way for broadening the application of remote sensing techniques across various disciplines.

The innovations in environmental and geological analysis have led to new research directions in software engineering. Atzberger et al. [23] made a significant contribution with their method of visualizing source code similarity through the use of 2.5D semantic software maps. Their method involved placing 3D glyphs on a two-dimensional plane to represent semantic relationships within source code, offering a novel method that facilitates enhanced code review and bug detection processes. Moreover, Khaloo et al. [24] introduced Code Park, a tool that reimagined codebase visualization as an interactive 3D environment. Innovations in software engineering visualization have led to intriguing developments, such as the method by Khaloo et al. [24], which transforms classes in a codebase into 3D rooms within Code Park. This method aims to make code comprehension more immersive, although it struggles with large, complex codebases, often resulting in cluttered visualizations. Likewise, Oberhauser et al. [25] proposed a 3D flythrough of code, which proves especially beneficial for newcomers to software projects. While this technique enhances various cognitive processes, it also encounters challenges when dealing with extensive codebases. Those three methods collectively broaden the capabilities of software engineers, each bringing its unique strengths to the field. Furthermore, the recent study on daylight harvesting in building design demonstrates the integration of remote sensing in urban development [26], using power over ethernet for efficient energy and sensor management. This aligns with our work, where we use remote-sensing-based terrain mapping to visualize Java source code in 3D, highlighting the versatility of remote sensing in complex system optimization.

Distinct from the methods mentioned above, we introduce a groundbreaking method for visualizing Java source code in 3D, utilizing terrain mapping techniques from remote sensing. We transform Java projects into vivid 3D forests, where each tree represents a distinct Java file, with attributes correlating to various code metrics. This visualization not only provides an intuitive view of the software system but also significantly aids in unraveling its structure and complexity. Our comprehensive multi-view analysis, incorporating project, file, and code component perspectives, enables a deeper, more nuanced understanding of the system's structure and complexity. Our method stands as a significant advancement in the field of software visualization, marking a notable leap forward in how software systems are understood.

3. Methodology

In our methodology, we employ a method that encompasses both the analysis of comprehensive datasets and the application of advanced visualization techniques. This section outlines the various steps and components of our methodology, starting with a detailed description of the datasets used in our study.

3.1. Dataset Description

The dataset [27] used for our study includes JDataMineMid and JDataMineLarge. According to the dataset's author, JDataMineMid encompasses over 300 Java projects, which include more than 80,000 Java files. JDataMineLarge is even larger, containing over 400 Java projects and more than 230,000 Java files. These files are derived from well-known projects on GitHub, each with a substantial number of stars, indicating their popularity and relevance in the developer community. This extensive and diverse collection of Java files provides a rich source of data, making it an ideal choice for our study.

3.2. Metric Selection for Different Perspectives

Appropriate metrics are a critical step in our method, as they directly influence the quality and relevance of the visualization. Given that our tool offers visualization analysis from three distinct perspectives-project view, file view, and code component view-, we have identified a set of metrics for each view that best capture their respective characteristics, as shown in Table 1.

Table 1. Metrics for project view, file view, and code component view.

Perspective	Metrics
Project View	Number of Java Files File Position in Project Structure
File View	Number of Methods Types of Member Properties Average Method Complexity
Component View	Method Name Token Count Cyclomatic Complexity Try-Catch Block Count Loop Count

- Users are provided with a high-level overview of the project's structure, visualized as a "code forest". Each Java file in the project is represented as a tree in this forest. The metrics we consider at this level include the number of Java files (or trees in the code forest) and the file position in the project structure. The number of Java files gives an understanding of the project's scale and complexity, while the position of a file in the project's structure is determined by the file's relative location in the project's directory structure or its position in the code forest.
- In the file-level, we delve into individual files, highlighting various attributes that provide insights into the file's nature and role within the project. The metrics we consider at this level include the number of methods, types of member properties, and average method complexity. The number of methods in a file indicates the file's responsibilities and complexity. The types of member properties reflect the different types present in a Java file. Lastly, the average method complexity metric gives a sense of the complexity of the methods in the corresponding Java file.
- In the code component-level, we focus on individual Java methods. The metrics considered here are the method name, token count, cyclomatic complexity, try-catch block count, and loop count. These metrics offer various perspectives on the complexity, size, and potential for errors within individual methods.

Collectively, these metrics offer a comprehensive view of the project's, file's, and method's characteristics, thereby facilitating a more informed and effective visualization.

3.3. Remote Sensing Inspired Code Visualization

Having established the overarching framework of layered structuring in the preceding section, we shift our focus to a thorough examination of the specific principles, formulas that constitute the core of our 3D visualization methodology. Our 3D code visualization tool, inspired by remote sensing techniques, provides users with three distinct perspectives to examine their projects: project-level, file-level, and code component-level.

3.3.1. Project-Level Construction

At the project-level, users are presented with a high-level overview of their project. The project's name and the number of Java files it contains are displayed in 3D text. Similar to how remote sensing visualizes geographical data, each Java file in the project is represented as a tree in a 3D forest. The position of each tree, which represents the hierarchical relationship of the files, is determined by the following formula:

$$P(T_i) = hash(F_i) \tag{1}$$

In Equation (1), $hash(F_i)$ is a function that maps the file's path. F_i represents the position in 3D coordinates, and $P(T_i)$ is the corresponding tree in the 3D forest visualization. This function is crucial for establishing the hierarchical relationship of the files in the project.

3.3.2. File-Level Construction

As users approach a tree, the tool transitions to the file-level, displaying detailed information about the corresponding Java file. At this level, different types of trees represent different types of class members, as shown in the figure below. From left to right, the trees represent member variables, regular methods, interfaces, and constructors.

Just as remote sensing uses color gradients to represent different data, the color of a tree in our visualization represents the average complexity of the methods in the corresponding Java file, with a gradient from green (low complexity) to yellow (high complexity). The complexity is determined by various factors, including cyclomatic complexity, documentation comments, code line count, the number of try-catch blocks, and the number of for-loops. The size of a tree is determined by the relative average line count of the methods in the file, with larger trees indicating more lines of code. The color of the trees transitions from green to yellow as the average complexity of the Java file.

$$S(T_i) = N(M_{F_i}) \tag{2}$$

$$C(T_i) = \frac{1}{N(M_{F_i})} \sum_{j=1}^{N(M_{F_i})} CC(M_j) \tag{3}$$

In Equations (2) and (3), $N(M_{F_i})$ is the number of methods in file F_i, and $CC(M_j)$ is the cyclomatic complexity of method M_j. The size of a tree, $S(T_i)$, is determined by the relative average line count of the methods in the file, and the color of a tree, $C(T_i)$, represents the average complexity of the methods in the corresponding Java file. These visual cues provide a quick understanding of the complexity and size of the Java files.

3.3.3. Code Component-Level Construction

Much like how remote sensing provides detailed data about specific geographical areas, the code component-level offers a meticulous breakdown of every individual method in a Java file. This perspective enhances the understanding of a method's complexity and structure, presenting detailed insights beyond those available at the file-level.

The method name is essentially the identifier of the method. This is essentially the identifier of the method, which can offer an initial clue about its function and purpose based on the naming conventions used.

The token count represents the count of individual components in the method. This represents the count of individual components in the method, such as variables, operators, and literals. A higher token count could imply a higher complexity of the method. The token count, represented as $T(M_j)$, is calculated with:

$$T(M_j) = CountTokens(M_j) \tag{4}$$

In Equation (4), $T(M_j)$ denotes the total number of tokens in method M_j, and "Count-Tokens" is the function used to calculate it.

The cyclomatic complexity represents the complexity of a program based on the number of linearly independent paths through the source code of the method. This represents the complexity of a program based on the number of linearly independent paths through the source code of the method. A high cyclomatic complexity suggests a method is more complex and potentially more challenging to maintain. The cyclomatic complexity, represented as $CC(M_j)$, is given by:

$$CC(M_j) = E - N + 2P \tag{5}$$

In Equation (5), E stands for the number of edges in the flow graph, N is the number of nodes, and P represents the number of connected components. To keep things simple, we won't detail each of the equations for these metrics. However, they include metrics such as volume, difficulty, and effort, offering a comprehensive perspective on the complexity of the software.

Overall, by offering these metrics, the code component-level allows users to delve into the specifics of each method, leading to a comprehensive understanding of its structure and complexity. This is particularly valuable for tasks such as code review, debugging, and maintenance, where a deep grasp of the method's behavior is crucial.

3.4. 3D Visualization of Datasets

Transforming our datasets into dynamic, interactive 3D visualizations was achieved through the use of "Three.js", which is a versatile tool that allows for the creation and rendering of complex 3D scenes directly in a web browser. The goal of these visualizations is to present the datasets in a manner that is both distinct and intuitive, moving beyond the limitations of traditional 2D representations. The initial phase of this transformation involved data preparation, where we extracted essential metrics from the datasets. These included the number of Java files per project, the count of methods in each file, and various metrics at the component-level. A combination of Java parsing libraries and custom scripts were employed to process these metrics, shaping them into a JSON structure readily interpretable.

In the next phase, "Three.js" was utilized to create individual 3D scenes for each type of visualization—project-level, file-level, and code component-level. Each scene, acting as a container, incorporated all the necessary elements, including objects, lights, and cameras. The 3D objects representing different elements of the datasets were crafted using a variety of geometries and materials to accurately depict different dataset characteristics. The positioning of these objects within each scene was meticulously determined by the dataset's underlying data, with trees in the project-level symbolizing Java files and positioned based on the file's path, while in the file and code component-levels, trees and blocks represented class members and methods.

To enhance the realism and depth of the 3D visualizations, lighting, and camera settings were carefully configured. A combination of ambient, directional, and point lights illuminated the objects and created shadows, enhancing the three-dimensional effect. Perspective cameras were set up to allow users to navigate the scenes and view objects from various angles. Finally, the rendering capabilities of “Three.js” brought these scenes to life within a web browser, complete with interactive features that enabled users to explore, zoom, and click on objects for more detailed information. This approach to 3D visualization provides a comprehensive and intuitive overview of the codebase, empowering developers to quickly identify key areas of interest or concern, and is particularly effective when combined with the detailed metrics available at the project, file, and method levels.

4. Evaluation

In this study, we apply our methodology to two real-world Java datasets, JDataMineMid and JDataMineLarge. Each dataset comprises a large number of Java projects, which in turn contain numerous Java files. These projects and files are derived from well-known projects on GitHub, each with a substantial number of stars, indicating their popularity and relevance in the developer community.

4.1. Project-Level Analysis

First, we conducted a size analysis of the project-level metric. To assess the efficacy of our method across Java projects with varying scales and complexities, we tallied the total number of Java files in each project. The corresponding outcomes are displayed in Table 2.

Table 2. Number of projects and Java files in two datasets.

Dataset	Number of Projects	Number of Java Files
JDataMineMid	301	86,526
JDataMineLarge	425	231,065

The JDataMineLarge dataset comprises the highest number of Java files, with 231,065 files, followed by JDataMineMid with 86,526 files. We observe a positive correlation where an increase in the total number of Java projects within a dataset is associated with an increase in the total number of Java files. For example, the JDataMineMid dataset, which has fewer projects, contains 86,526 Java files, while the JDataMineLarge dataset, with more projects, comprises 231,065 Java files.

4.2. File-Level Analysis

In this study, we undertook a comprehensive analysis of effective documentation comments within two substantial Java datasets, namely JDataMineMid and JDataMineLarge. We define effective documentation comments as those that either start with the syntax “/**” or “/*” and encompass at least one line of substantive content. It is important to note that our analysis specifically excludes single-line comments that begin with “//”, as these are often less formal and may not provide substantial documentation.

Our findings offer valuable insights into the practices of Java documentation. We discovered a notable prevalence of effective documentation comments in Java codebases. Specifically, a significant majority of the Java files in both datasets exhibit this form of documentation. In JDataMineMid, 65.5% of the analyzed files contain effective documentation comments, while in JDataMineLarge, this figure rises to 82.0%. These results underscore the importance and widespread use of detailed commenting in Java projects for enhancing code readability and maintenance. The detailed breakdown of these findings is presented in Table 3.

Table 3. Number of Java files with effective documentation comments in codebases.

Dataset	Files with Documentation	Percentage
JDataMineMid	56,669	65.5%
JDataMineLarge	189,534	82.0%

4.3. Component-Level Analysis

This experiment primarily investigates the complexity of code within the JDataMine-Mid and JDataMineLarge code repositories, focusing on structural elements like the number of try-catch blocks and for-loop iterations. Control flow structures such as loops and try-catch blocks are fundamental constructs in Java programming, where their presence and frequency can significantly impact both the complexity and readability of the code. For loops, introducing a level of iteration and repetition can increase a method's complexity. Similarly, try-catch blocks, crucial for exception handling, often mark areas in the code where exceptions are anticipated and handled, adding to the method's complexity.

Alongside these structural elements, we also delve into the lexical aspect of code complexity by examining the distribution of method name token lengths. We analyze how programmers tend to construct method names, which is indicative of preferences in coding style and clarity. Our findings, based on the token count of method names, are visually represented in Figure 2. The figure illustrates that method names with two tokens are most common in all datasets, while those with eight or more tokens are comparatively rare. This distribution suggests a general tendency among programmers to favor clear and concise method names, opting for brevity over verbosity in naming conventions.

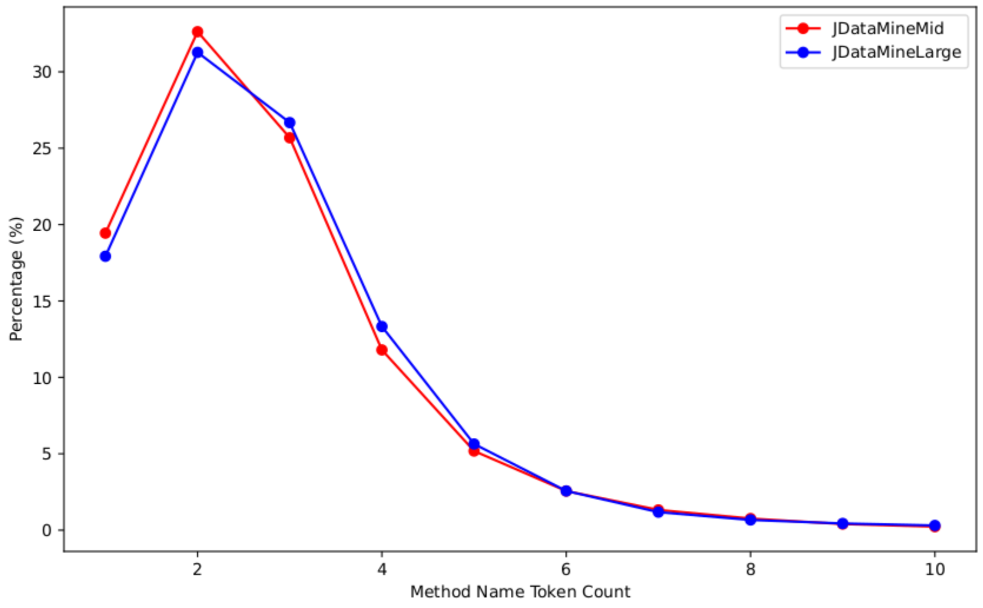
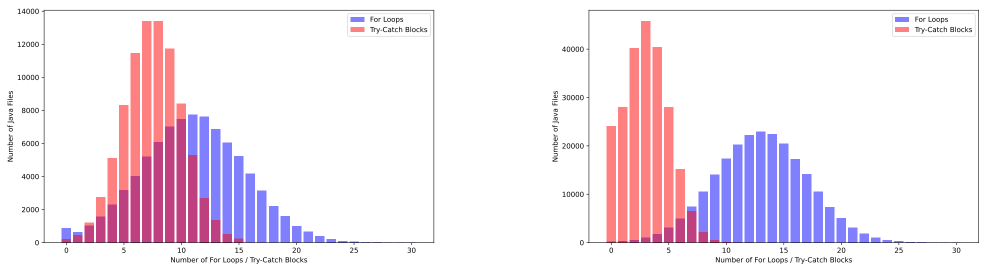


Figure 2. Distribution of method name token length.

In this subsection, we analyze the number of Java methods in our datasets that contain loops and try-catch blocks. This analysis provides insights into the complexity of the methods in the datasets and serves as a complement to the cyclomatic complexity analysis, offering a more nuanced view of method complexity.

Figure 3a illustrates the distribution of for loops and try-catch blocks across the Java files in the JDataMineMid dataset. The x-axis represents the number of loops or try-catch blocks, while the y-axis represents the number of Java files. This figure shows that the majority of Java files have a moderate number of loops and try-catch blocks, with a few outliers having a high number of these constructs.

Similarly, Figure 3b depicts the distribution of for loops and try-catch blocks in the JDataMineLarge dataset. The distribution in this larger dataset follows a similar pattern to that of the JDataMineMid dataset, reinforcing the observations made from the smaller dataset.



(**a**) (**b**)

Figure 3. Distribution of "for loops" and try-catch blocks across Java files in the JDataMineMid dataset and the JDataMineLarge dataset: (**a**) Distribution in the JDataMineMid dataset; (**b**) Distribution in the JDataMineLarge dataset.

These analyses provide a comprehensive view of the complexity of Java methods in our datasets, considering the presence of key control flow constructs. These two metrics can effectively reflect the overall complexity of the code.

In this section, we describe the methodology used to collect and filter the raw data obtained from GitHub, resulting in three high-quality datasets. The experimental process was divided into three parts: data acquisition, data statistics, and dataset scoring.

4.4. Visualization Experiments

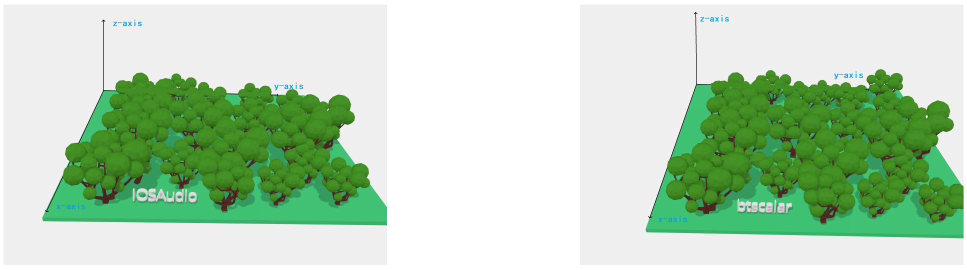
In this section, we present the results of applying our 3D visualization methodology to the JDataMineMid and JDataMineLarge datasets. We provide a detailed description of the 3D models generated for each dataset, highlighting the insights that can be gained from these visualizations.

4.4.1. Project-Level Visualization

Our approach commenced with the application of our innovative methodology at the project-level, where we created a comprehensive 3D model for a representative project from each of the datasets, JDataMineMid and JDataMineLarge. This step was instrumental in providing a macroscopic view of each project's structure and complexity. In the resultant 3D models, we uniquely represented each Java file as a tree within a virtual 3D forest. The innovative aspect here is the depiction of each Java file's size by the corresponding tree's size, offering a visually intuitive measure of the file's magnitude. Furthermore, the placement of each tree in this virtual forest was meticulously calculated using a hash function. This function maps the file's path to a specific 3D coordinate, thus ensuring that each file is consistently and uniquely positioned within the model.

Another key feature of our visualization technique is the incorporation of the project's name in 3D text within the model, enhancing the ease of identification of each project within its forest representation. This integration of text and 3D modeling provides a clear and immediate reference point for viewers.

The visualizations effectively translate the architectural complexity of a software project into an understandable and visually engaging format. The spatial arrangement and distribution of trees in these models offer insights into the project's file organization and overall structure, converting the abstract concept of software architecture into a tangible visual experience. The visual impact of these project-level visualizations for a selected project from both the JDataMineMid and JDataMineLarge datasets can be fully appreciated in Figure 4a,b, where the forest of trees metaphorically stands for the Java files, laying out the structural blueprint of the projects.



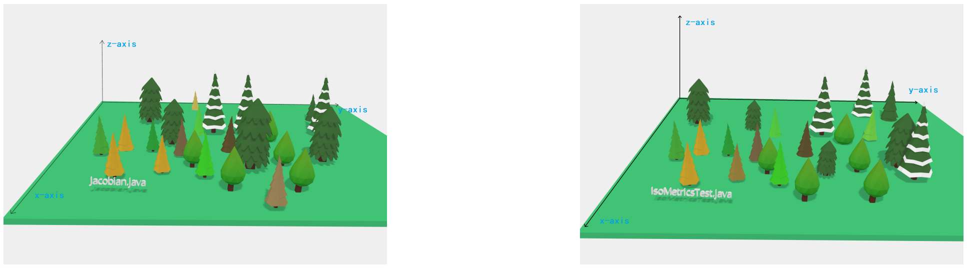
(**a**) (**b**)

Figure 4. Code forest visualizations: (**a**) A selected project from the JDataMineMid dataset; (**b**) A selected project from the JDataMineLarge dataset.

4.4.2. File-Level Visualization

Progressing into the subsequent stage of our research, we amplified our methodology to encompass the file-level, engendering unique 3D models for selected Java files within the datasets. Each file is portrayed as a unique landscape, where diverse tree types symbolize varying types of class members contained within the file. The color of each tree is dictated by the average complexity of the methods in the associated Java file, with diverse colors representing different complexity levels. The size of each tree symbolizes the relative average line count of the methods in the file, providing a visually apparent measure of the method's size. The position of each tree is determined by the sequence of the class members within the file, ensuring a coherent and consistent layout.

Figure 5 depicts the file-level visualizations for the "jacobian.java" file from the JDataMineMid dataset and the "isometrictileTest.java" file from the JDataMineLarge dataset, respectively. These figures vividly portray various types of trees representing diverse class members in each Java file. The color and size of the trees provide a visually engaging representation of the complexity and size of the methods in each file, with different colors illustrating varying levels of method complexity.



(**a**) (**b**)

Figure 5. File-level visualizations in the JDataMineMid and JDataMineLarge datasets: (**a**) File-level visualization of the "jacobian.java" file from the JDataMineMid dataset; (**b**) File-level visualization of the "isoMetricsTest.java" file from the JDataMineLarge dataset.

Our 3D visualization approach at the file-level offers a distinctive lens to view the structure and complexity of individual Java files. By transmuting each file into a 3D landscape, we introduce a more intuitive and engaging mode to comprehend the file's contents. This could assist software engineers in swiftly identifying areas of interest or concern, such as intricate methods or voluminous class members, thereby enabling more informed decisions about code maintenance and optimization.

Additionally, our file-level visualization can also shed light on the overall structure and organization of the Java files in the datasets. By contrasting the 3D models of different files, we can discern patterns and trends in the distribution of class members and method complexity. This could offer valuable insights into the coding practices and conventions adopted in the datasets, potentially influencing future software development practices.

In conclusion, our file-level visualization approach provides a powerful instrument for comprehending and analyzing Java files. By embodying each file as a 3D landscape, we can unearth insights that might not be immediately evident from the 2D textual depiction of the file. This could prove invaluable for tasks such as code review, debugging, and maintenance, where a profound understanding of the file's contents is crucial.

4.4.3. Component-Level Visualization

In the final phase of our research, we concentrated on a more detailed view of analysis, specifically focusing on individual methods within the datasets. By creating unique 3D models for each method, we were able to deliver a comprehensive and interactive representation of component-level attributes, offering an in-depth perspective on the intricacies of each method.

Our investigation included a detailed exploration of method name token lengths in the JDataMineMid dataset. We discovered a clear trend indicating a preference for brevity and simplicity in method naming conventions. Specifically, we observed that method names consisting of two tokens are the most prevalent within this dataset, highlighting a general tendency toward concise naming. Conversely, method names composed of seven or more tokens are significantly less frequent. This pattern underscores the inclination of programmers to avoid overly complex or verbose method names. The nuances of these findings are vividly depicted in the 3D visualization shown in Figure 6, where the distribution of method name token lengths is clearly illustrated, providing visual evidence of these naming trends.

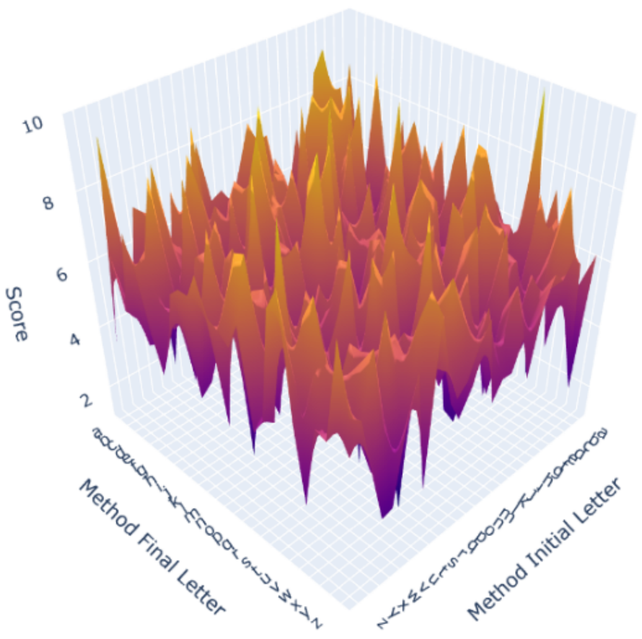


Figure 6. 3D visualization of method name token length distribution in the JDataMineMid dataset.

In our study, we first focused on examining the distribution of various method types within the JDataMineMid dataset. This examination led to the creation of a 3D pie chart, as depicted in Figure 7, which highlights the predominance of different methods. Notably, the chart reveals that 'get' methods form the largest segment, making up nearly one-fifth of all methods. This finding provides critical insight into the prevalent practices of method implementation within this dataset.

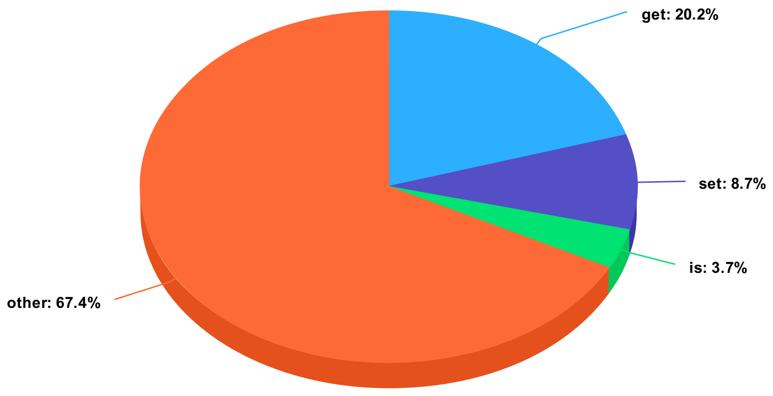


Figure 7. Proportion of get, set, is, and other methods in the JDataMineMid dataset.

Following this, we delved deeper into the data with a detailed 3D heatmap analysis. Our heatmap serves as a nuanced visual representation, with the x-axis labeled from ’a’ to ’z’ representing the first letter of method names, and the y-axis, similarly labeled, representing the last letter. This categorizes the methods into 676 distinct groups based on the combination of their initial and final letters. For each of these groups, we calculated average scores using our predefined metrics. The heatmap’s height and color gradients, which resemble a topographic map generated through remote sensing, represent these scores. This visual tool not only provides an intuitive understanding of the performance of each category but also facilitates a comprehensive overview and comparative analysis across the different groups. This analysis is instrumental in yielding invaluable insights into method naming trends and their broader implications within the dataset.

It is pertinent to highlight the distinctive features of our methodology in comparison to other existing approaches. Table 4 provides a concise comparison between our method and “Code Park” [24], underscoring the unique aspects of our approach. Specifically, our methodology employs a three-layered structural visualization using Java, contrasting sharply with “Code Park”’s C# blackboard-style presentation. The intuitive and multi-perspective nature of our visualization, particularly at the component level, offers a more profound and interactive understanding of code structures.

Table 4. Comparison between Our Method and Code Park.

Feature	Our Method	Code Park
Programming Language	Java	C#
Structural Layering	3 layers: Project, File, Component	None
Visualization Style	Trees representing code structure	Source code on a blackboard
Understandability	Easy	Normal

In summary, our 3D visualization methodology provides a potent tool for code comprehension and analysis. By manifesting code in 3D, we can unveil insights that might not be immediately perceptible from the 2D textual depiction of the code. This could prove to be invaluable for tasks such as code review, debugging, and maintenance, where a profound understanding of the code is paramount. Our component-level visualization, in particular, presents a detailed and interactive perspective of individual methods, enriching our comprehension of their attributes and their role within the larger codebase.

5. Conclusions and Future Directions

This research represents a significant advancement in the integration of remote sensing technology with software engineering, with a specific focus on the visualization of Java code data. We have developed a pioneering method for the 3D visualization of Java source code files, markedly enhancing the ability of software engineers to comprehend and analyze code. This approach not only simplifies complex data but also enriches the interpretation of Java code, providing a more intuitive understanding for software engineers. By preprocessing and transforming code data and applying intelligent techniques, we have unlocked crucial insights into code complexity, structure, and potential issues. These insights are expected to streamline software development processes and improve code quality. However, our approach has limitations that offer opportunities for future research.

However, one aspect that requires further refinement is the management of visualization density in larger and more complex codebases. This aspect can occasionally make it challenging to efficiently extract meaningful insights about code quality, complexity, and structure. Our future research aims to refine visualization methods to better handle larger and more intricate codebases, potentially through transformation techniques, and innovative visualization models. We also plan to extend our research to include more programming languages beyond Java, enhancing the versatility of our approach. We will employ more remote sensing technologies to address software engineering challenges, such as aiding in energy monitoring and sustainability. Another focus will be on improving the interactivity of our visualizations by incorporating advanced search functionality, filters, and real-time updates to enrich the user experience. Our method is also planned to be utilized for assessing its usability, efficacy, and impact on software development. In summary, our research has made significant contributions to software engineering, and we are excited about continuing to explore and innovate in the realm of intelligent visualization of code data, pushing the boundaries of current methodologies.

Author Contributions: Conceptualization, D.L.; Methodology, D.L.; Software, D.L.; Validation, D.L. and Y.Z.; Formal analysis, D.L., W.W. and Y.Z.; Investigation, D.L., W.W. and Y.Z.; Resources, D.L., W.W. and Y.Z.; Data curation, D.L. and Y.Z.; Writing—original draft, D.L., W.W. and Y.Z.; Writing—review & editing, D.L. and W.W.; Visualization, W.W. and Y.Z.; Supervision, W.W.; Project administration, W.W.; Funding acquisition, W.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data used in this study are available from the authors upon request.

Acknowledgments: The authors would like to thank the anonymous reviewers for their constructive comments, which have significantly improved the quality of the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ji, Z.; Song, X.; Feng, Q.; Wang, H.; Chen, C.H.; Chang, C.C. RSG-Net: A Recurrent Similarity Network With Ghost Convolution for Wheelset Laser Stripe Image Inpainting. *IEEE Trans. Intell. Transp. Syst.* **2022**, *24*, 12852–12861. [CrossRef]
2. Guo, X.; Ji, Z.; Feng, Q.; Wang, H.; Yang, Y.; Li, Z. URS: A Light-Weight Segmentation Model for Train Wheelset Monitoring. *IEEE Trans. Intell. Transp. Syst.* **2022**, *24*, 7707–7716. [CrossRef]
3. da Silva, C.A.G.; de Sá, J.L.R.; Menegatti, R. Diagnostic of failure in transmission system of agriculture tractors using predictive maintenance based software. *AgriEngineering* **2019**, *1*, 132–144. [CrossRef]
4. Ikram, A.; Jalil, M.A.; Ngah, A.B.; Khan, A.S.; Mahmood, Y. An Empirical Investigation of Vendor Readiness to Assess Offshore Software Maintenance Outsourcing Project. *Int. Comput. Sci. Netw. Secur.* **2022**, *22*, 229.
5. Cerny, T.; Abdelfattah, A.S.; Bushong, V.; Al Maruf, A.; Taibi, D. Microservice architecture reconstruction and visualization techniques: A review. In Proceedings of the 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE), Newark, CA, USA, 15–18 August 2022; pp. 39–48.
6. Enrici, A.; Apvrille, L.; Pacalet, R.; Pham, M.H. Static Data-Flow Analysis of UML/SysML Functional Views for Signal and Image Processing Applications. In Proceedings of the Model-Driven Engineering and Software Development: 7th International Conference, MODELSWARD 2019, Prague, Czech Republic, 20–22 February 2019; pp. 101–126.

7. Khanal, S.; Kc, K.; Fulton, J.P.; Shearer, S.; Ozkan, E. Remote sensing in agriculture—Accomplishments, limitations, and opportunities. *Remote Sens.* **2020**, *12*, 3783. [CrossRef]
8. Weiss, M.; Jacob, F.; Duveiller, G. Remote sensing for agricultural applications: A meta-review. *Remote Sens. Environ.* **2020**, *236*, 111402. [CrossRef]
9. Lechner, A.M.; Foody, G.M.; Boyd, D.S. Applications in remote sensing to forest ecology and management. *ONE Earth* **2020**, *2*, 405–412. [CrossRef]
10. Li, Z.Z.; Li, Z.C.; Sun, Y.R.; Ahmad, H.; Xu, G.; Chen, X.B. Quantum Remote State Preparation Based on Quantum Network Coding. *Comput. Mater. Contin.* **2022**, *73*, 119–132. [CrossRef]
11. Khelifi, L.; Mignotte, M. Deep learning for change detection in remote sensing images: Comprehensive review and meta-analysis. *IEEE Access* **2020**, *8*, 126385–126400. [CrossRef]
12. Alexandropoulos, S.A.N.; Kotsiantis, S.B.; Vrahatis, M.N. Data preprocessing in predictive data mining. *Knowl. Eng. Rev.* **2019**, *34*, e1. [CrossRef]
13. Luengo, J.; García-Gil, D.; Ramírez-Gallego, S.; García, S.; Herrera, F. *Big Data Preprocessing*; Springer: Cham, Switzerland, 2020.
14. Sullivan, C.; Kaszynski, A. PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK). *J. Open Source Softw.* **2019**, *4*, 1450. [CrossRef]
15. Draz, M.M.; Farhan, M.S.; Abdulkader, S.N.; Gafar, M. Code smell detection using whale optimization algorithm. *Comput. Mater. Contin.* **2021**, *68*, 1919–1935. [CrossRef]
16. Moral-Muñoz, J.A.; Herrera-Viedma, E.; Santisteban-Espejo, A.; Cobo, M.J. Software tools for conducting bibliometric analysis in science: An up-to-date review. *Prof. Inf.* **2020**, *29*, 4. [CrossRef]
17. Lourenço, P.; Teodoro, A.C.; Gonçalves, J.A.; Honrado, J.P.; Cunha, M.; Sillero, N. Assessing the performance of different OBIA software approaches for mapping invasive alien plants along roads with remote sensing data. *Int. J. Appl. Earth Obs. Geoinf.* **2021**, *95*, 102263. [CrossRef]
18. Quattrochi, D.A.; Lam, N.S.N.; Qiu, H.L.; Zhao, W. Image characterization and modeling system (ICAMS): A geographic information system for the characterization and modeling of multiscale remote sensing data. In *Scale in Remote Sensing and GIS*; Routledge: London, UK, 2023; pp. 295–307.
19. Congedo, L. Semi-Automatic Classification Plugin: A Python tool for the download and processing of remote sensing images in QGIS. *J. Open Source Softw.* **2021**, *6*, 3172. [CrossRef]
20. Lindberg, E.; Holmgren, J. Individual tree crown methods for 3D data from remote sensing. *Curr. For. Rep.* **2017**, *3*, 19–31. [CrossRef]
21. Mikita, T.; Balková, M.; Bajer, A.; Cibulka, M.; Patočka, Z. Comparison of different remote sensing methods for 3d modeling of small rock outcrops. *Sensors* **2020**, *20*, 1663. [CrossRef] [PubMed]
22. Qi, J.; Xie, D.; Yin, T.; Yan, G.; Gastellu-Etchegorry, J.P.; Li, L.; Zhang, W.; Mu, X.; Norford, L.K. LESS: LargE-Scale remote sensing data and image simulation framework over heterogeneous 3D scenes. *Remote Sens. Environ.* **2019**, *221*, 695–706. [CrossRef]
23. Atzberger, D.; Cech, T.; Scheibel, W.; Limberger, D.; Döllner, J. Visualization of Source Code Similarity Using 2.5 D Semantic Software Maps. In Proceedings of the International Joint Conference on Computer Vision, Imaging and Computer Graphics, Valletta, Malta, 27–29 February 2021; pp. 162–182.
24. Khaloo, P.; Maghoumi, M.; Taranta, E.; Bettner, D.; Laviola, J. Code park: A new 3d code visualization tool. In Proceedings of the 2017 IEEE Working Conference on Software Visualization (VISSOFT), Shanghai, China, 18–19 September 2017; pp. 43–53.
25. Oberhauser, R.; Silfang, C.; Lecon, C. Code structure visualization using 3D-flythrough. In Proceedings of the 2016 11th International Conference on Computer Science & Education (ICCSE), Nagoya, Japan, 23–25 August 2016; pp. 365–370.
26. Kent, M.; Huynh, N.K.; Schiavon, S.; Selkowitz, S. Using support vector machine to detect desk illuminance sensor blockage for closed-loop daylight harvesting. *Energy Build.* **2022**, *274*, 112443. [CrossRef]
27. Li, D.; Wang, W.; Zhao, Y. Intelligent Java Dataset Construction and Visualization Evaluation for Reliable Software Development. In Proceedings of the 2023 International Conference on Frontiers of Robotics and Software Engineering (FRSE), Changsha, China, 17–18 March 2023; pp. 263–270.